# Packing sub-word-size arguments

Zoltan Somogyi
YesLogic

YesLogic
2018 October 12

# Term representation

On a 64 bit architecture, aligned pointers always have the bottom three bits clear.

Mercury has always used those bottom bits as a *primary tag*.

In the simplest case, every function symbol gets its own ptag value. We add the ptag value to the pointer to the memory cell containing the arguments; we subtract it, or mask it off, before following the pointer.

This works for up to eight function symbols.

```
:- type t
    --->    f0(...)   % ptag 0
    ;       f1(...)   % ptag 1
    ;       ...
    ;       f6(...)   % ptag 6
    ;       f7(...).  % ptag 7
```

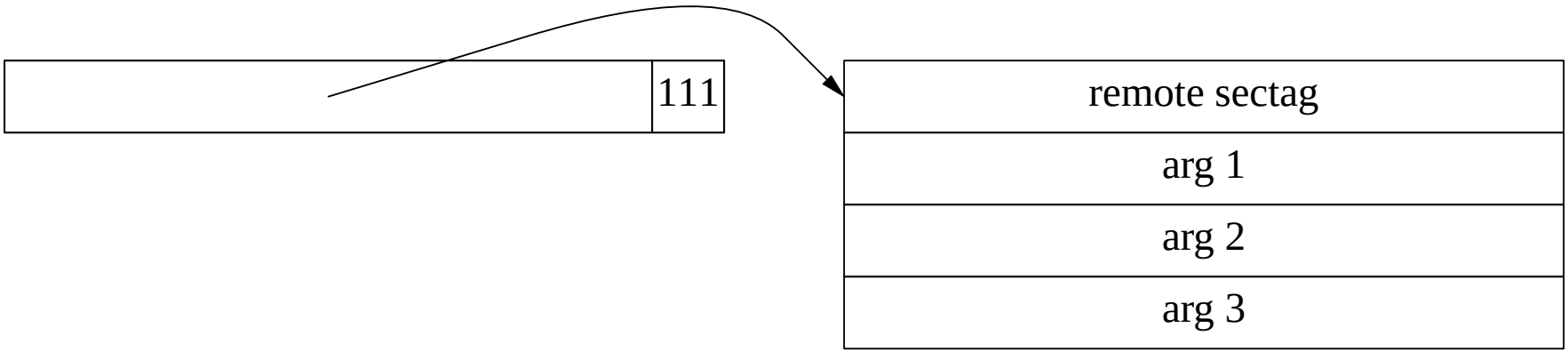
# Local secondary tags

Constants, i.e. function symbols without arguments, don't need a memory cell. We allocate the ptag value 0 to all constants in a type, and distinguish between them using a *local secondary tag* stored in the 61 bits left over.

```
:- type t
    --->    c0          % ptag 0, local sectag 0
    ;       c1          % ptag 0, local sectag 1
    ;       c2          % ptag 0, local sectag 2
    ;       f1(...)     % ptag 1
    ;       ...
```

# Remote secondary tags

- We assign ptag value 0 to all the constants, if any.
- We assign ptag values up to 6 to nonconstant function symbols.
- If there is more than one function symbol remaining, then we assign the last ptag value (7) to *all* of them, and distinguish between them using a 64 bit *remote secondary tag* stored in the first word of the memory cell.



# Types using other allocation schemes

We represent values of types containing only N constants using the integers 0 .. N-1, like enums in C.

```
:- type suit ---> club ; diamond ; heart ; spades.
```

Types with one function symbol of arity one are notag types. We represent them as if the wrapper wasn't there, e.g. we represent a counter as an int.

```
:- type counter ---> counter(int).
```

Types with one function symbol of arity zero are dummy types. Their representation does not need any bits.

```
:- type dummy ---> dummy.
```

# Direct arg optimization

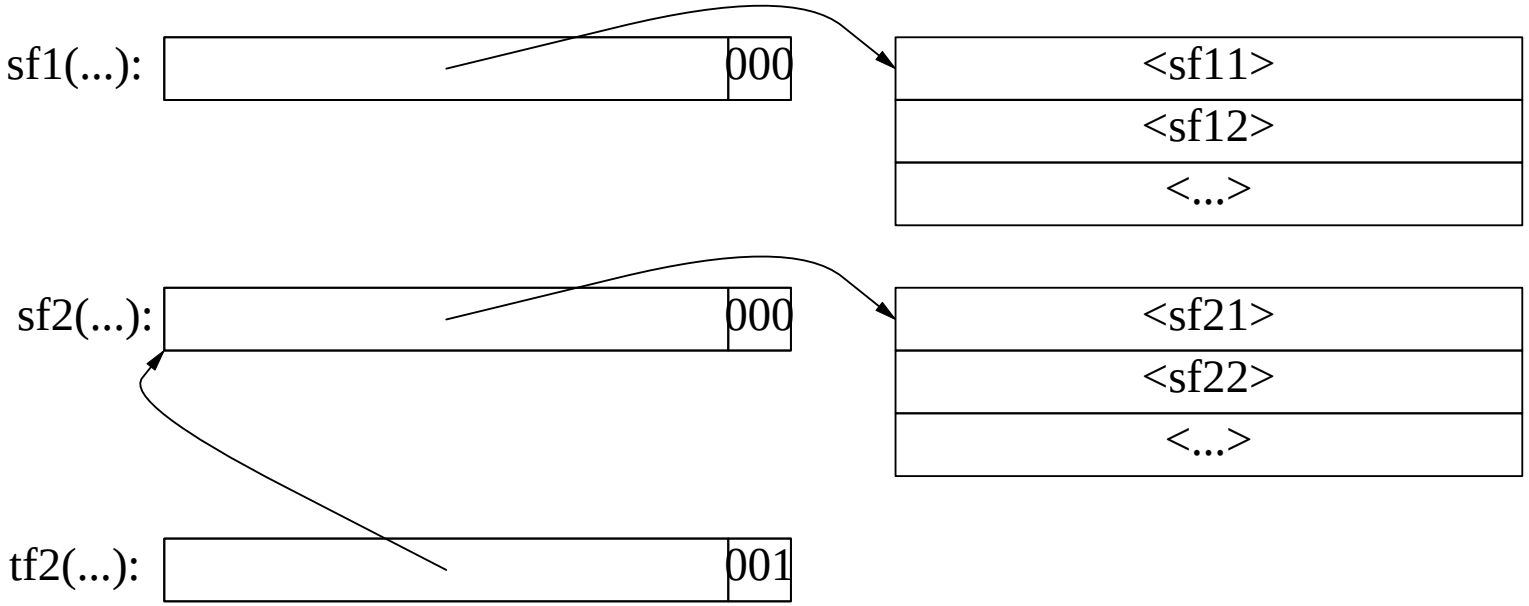
This optimization applies to function symbols with exactly one argument (such as `tf1` and `tf2`) where that argument is of a type whose values' ptags are guaranteed to be 000 (such as `s1` and `s2`).

```
:- type s1 ---> sf1(sf11, sf12, ...).
:- type s2 ---> sf2(sf21, sf22, ...).

:- type t  ---> tf1(s1)
           ;    tf2(s2).
```

The optimization repurposes those guaranteed-to-be-000 bits in `s1` and `s2` to distinguish between `tf1` and `tf2`.

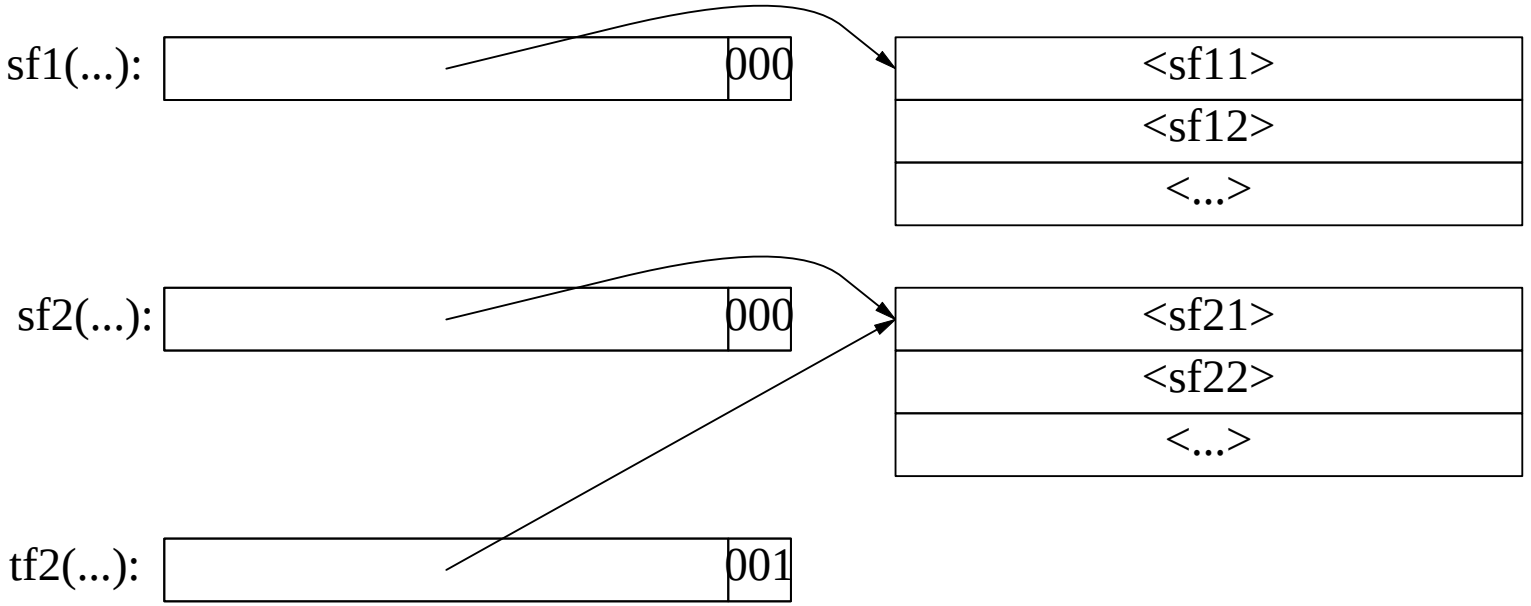
# Without direct arg optimization

```
:- type s1 ---> sf1(sf11, sf12, ...).
:- type s2 ---> sf2(sf21, sf22, ...).

:- type t  ---> tf1(s1)
           ;    tf2(s2).
```



# With direct arg optimization

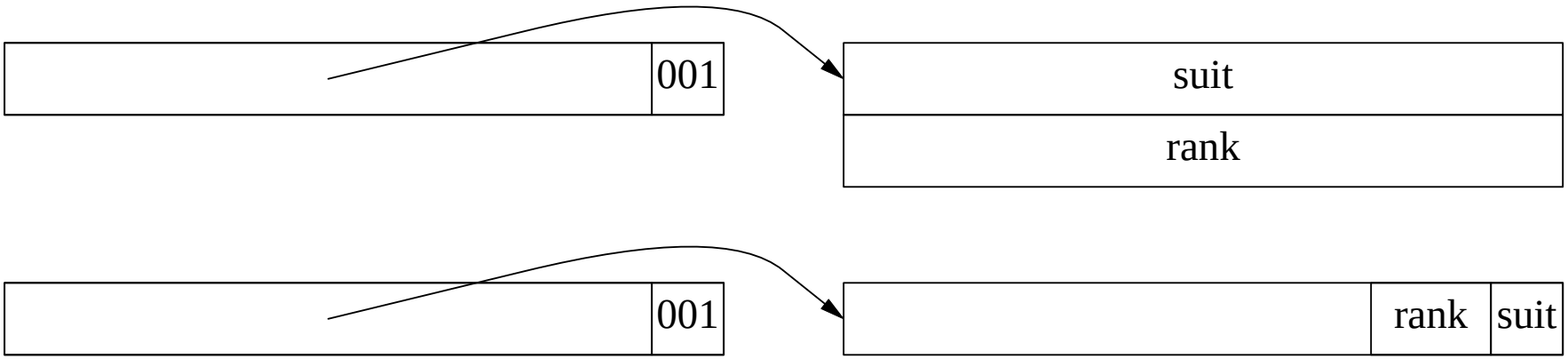
```
:- type s1 ---> sf1(sf11, sf12, ...).
:- type s2 ---> sf2(sf21, sf22, ...).

:- type t  ---> tf1(s1)
           ;    tf2(s2).
```



# Packing enums

Until 2011, all arguments were stored in one word in the heap cell (with the exception of floats on 32 bit systems, which used two words). Then Peter implemented packing of enums.

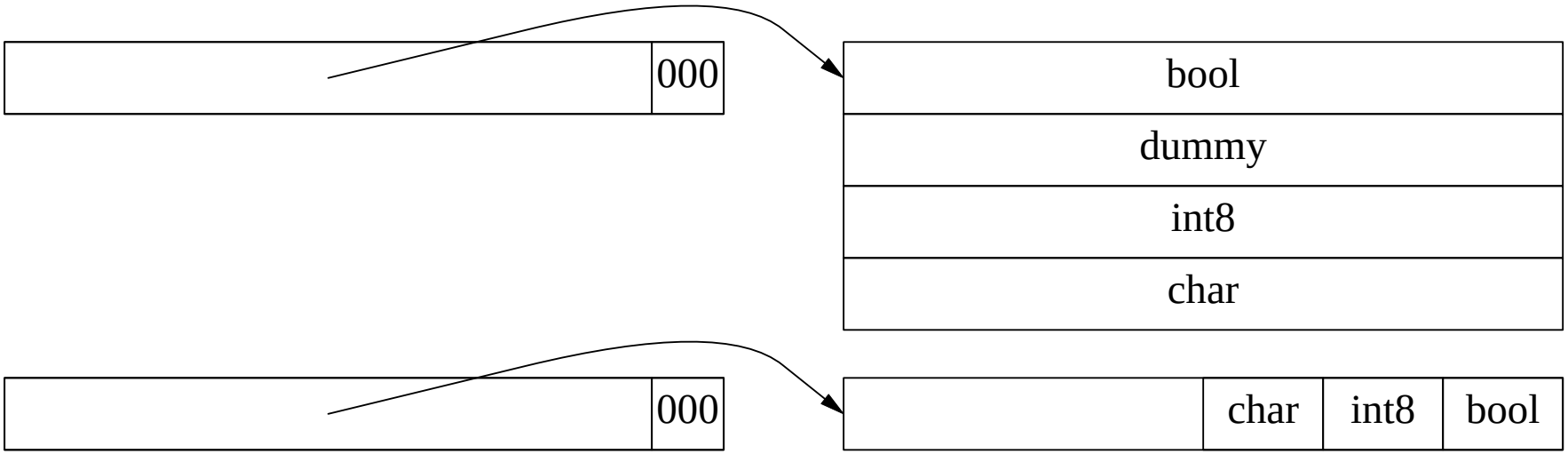
```
:- type suit ---> club ; diamond ; heart ; spades. % 2 bits
:- type rank ---> two ; three ; ... ; king ; ace.  % 4 bits
:- type card ---> card(suit, rank) ; joker.
```



# Packing non-enums

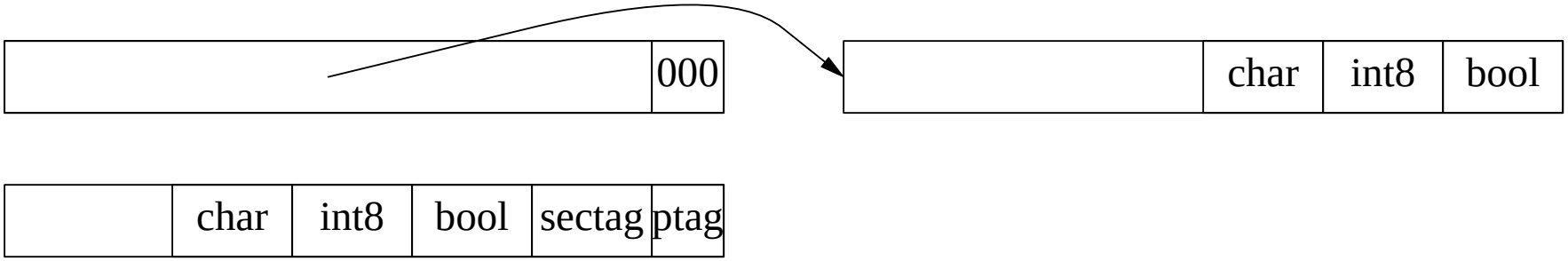
I extended Peter's work to also pack

- int{8,16,32} and uint{8,16,32} values
- Unicode chars (21 bits)
- values of dummy types (0 bits)



# Storing args next to local sectags

When the arguments of a function symbol fit together into one word, we may not need a separate memory cell at all.



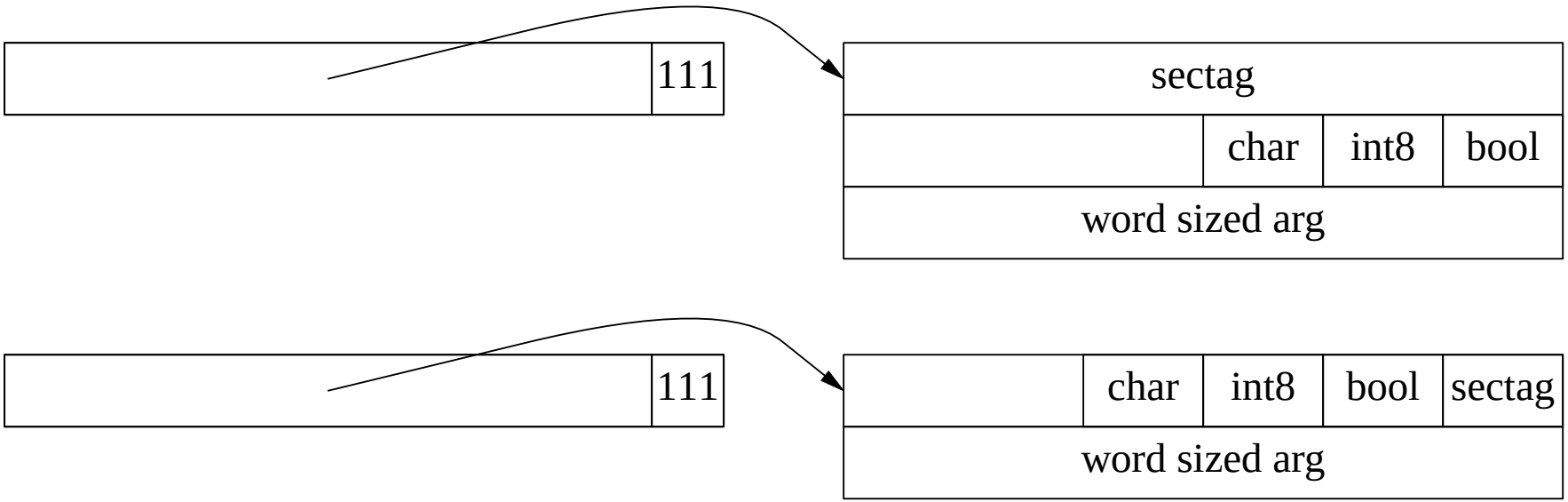

If a type has $N$ such function symbols where $N \leq 2^k$, we can apply this optimization to a function symbol if its arguments need at most $64 - 3 - k$ bits. (3 for primary tag, $k \geq 0$ for local secondary tag.)

We prefer to use this scheme for function symbols whose args are packable into fewer bits. This allows $k$ to be bigger, which allows more function symbols to be optimized this way.

# Storing args next to remote sectags

Remote secondary tags don't need to take up a full word either: they need only as many bits as is needed to differentiate the function symbols sharing the primary tag value.

We can use the remaining bits to store any initial subsequence of sub-word-sized arguments.



# Argument reordering

At the moment, the compiler can pack sub-word-sized arguments together into a word only when those arguments are next to each other. We can thus pack the two bools in `t2`, but not in `t1`.

```
:- type t1 ---> t1f(bool, map(...), bool, ...).
:- type t2 ---> t1f(bool, bool, map(...), ...).
```

With the right option, the compiler *will* generate an informational message telling the programmer that replacing `t1` with `t2` would be a win, and why.

It would be nice if the compiler could just *automatically* treat `t1` as it were written as `t2`.

The challenge is ensuring consistency across module boundaries when the definitions of some argument types are hidden behind abstraction barriers.

# Mini types

At the moment, for the compiler to recognize that a type is sub-word-sized and therefore packable, the type must be either

- an enum type,
- a dummy type, or
- a builtin type.

We should also recognize as packable

- notag types wrapped around packable types, such as `nt`, and
- general du types that can be stored in a part of a word, such as `t1`.

```
:- type nt ---> nt(int8).

:- type t1 ---> t1f(bool, int8). % can optimize now
:- type t2 ---> t2f(bool, nt).   % cannot optimize now
:- type t3 ---> t3f(bool, t1).   % cannot optimize now
```

# Mini types and module boundaries

Module boundaries are also the challenge for implementing mini types.

```
module A:                          module B:
:- type a1 ---> a1(bool).          :- type b1 ---> b1(a1).
:- type a2 ---> a2(b1, bool).      :- type b2 ---> b2(a2).
...
:- type a9 ---> a9(b8, bool).      :- type b9 ---> b9(a9).
```

To find the size of e.g. `a9`,

- we could do a fixpoint iteration, with each compiler invocation adding information about one more type to either A's or B's interface file, or
- we could put constraints such as `sizeof(a9) = sizeof(b8) + 1` into interface files, and let the readers of those interface files solve the constraints.

# Updating non-packed args

```
p(T0, T) :-
    T0 = f1(A, B, C, _, E, F, G),
    D = ...,
    T  = f1(A, B, C, D, E, F, G).
```

We used to translate this to code that picks up A,B,C,E,F,G from T0 separately, and then packs them all up to generate T, even when both the first three and last three args are packed together.

We now save many shifts, ANDs and ORs in hlc grades by generating

```
MR_Unsigned packed_word_0 = field(mktag(0), T0, 0);
MR_Unsigned packed_word_1 = field(mktag(0), T0, 2);
T = alloc(...);
field(mktag(0), T, 0) = packed_word_0;
field(mktag(0), T, 1) = D;
field(mktag(0), T, 2) = packed_word_1;
```

# Updating packed args

We can reuse *part* of a packed word as well.

```
p(T0, T) :-
    T0 = f1(A, B, _, D, E, F, G),
    C = ...,
    T  = f1(A, B, C, D, E, F, G).
```

```
MR_Unsigned packed_word_0 = field(mktag(0), T0, 0);
D =                         field(mktag(0), T0, 1);
MR_Unsigned packed_word_1 = field(mktag(0), T0, 2);
T = alloc(...);
field(mktag(0), T, 0) = (packed_word_0 & (~0x...)) | C;
field(mktag(0), T, 1) = D;
field(mktag(0), T, 2) = packed_word_1;
```

# Automatically generated unify predicate

The compiler has always generated a type-specific unification predicate for every type constructor. This predicate

- figures out what function symbol the args X and Y are bound to,
- requires the two function symbols to be the same, then
- pairwise unifies the function symbols' corresponding arguments.

```
__Unify__(X, Y) :-
  (
    X = f(XF1, XF2, ...), Y = f(YF1, YF2, ...),
    XF1 = YF1, XF2 = YF2, ...
  ;
    ...
  ).
```

The argument unifications will turn into simple tests for atomic types, and calls to type-specific unification predicates for nonatomic types.

# Unifying packed arguments

We initially kept unifying arguments one by one even if e.g. the first two are packed together, leading to code like this

```
  XF1 = field(mktag(0), X, 0) & 3;
  YF1 = field(mktag(0), Y, 0) & 3;
  XF2 = (field(mktag(0), X, 0) >> 2) & 3);
  YF2 = (field(mktag(0), Y, 0) >> 2) & 3);
  ... pick up later args in other words ...
  succeeded = (XF1 == YF1);
  if (succeeded) {
    succeeded = (XF2 == YF2);
    if (succeeded) {
      ... compare later args
    }
  }
}
```

# Unify in bulk

There is no point in picking apart the word containing `f`'s first two arguments. We can compare those words from `X` and `Y` in their entirety, yielding shorter, faster code. (The bits that do not store arguments are guaranteed to be zero.)

```
  WordX1 = field(mktag(0), X, 0);
  WordY1 = field(mktag(0), X, 0);
  ... pick up later args in other words ...
  succeeded = (WordX1 == WordY1);
  if (succeeded) {
    ... compare later args
  }
}
```

# Automatically generated compare predicate

The compiler has always generated a type-specific comparison predicate for every type constructor as well. This predicate

- figures out what function symbol X and Y are bound to,
- return the result if they are different, and otherwise
- pairwise compares the function symbol's arguments in turn.

The code for the last part has always looked like this:

```
X = f(XF1, XF2, ...),
Y = f(YF1, YF2, ...),
compare(R1, XF1, YF1),
( if R1 != (=) then
  R = R1
else
  ...
)
```

# Compare in bulk

When an argument word contains two or more sub-word-sized arguments packed together, we can compare this word from `X` and `Y` all at once, provided

- the arguments are stored in an order that puts earlier arguments into more significant bit positions, and
- all the arguments compare as unsigned.

```
:- type b ---> b0 ; b1.
:- type oct ---> oct(b, b, b).
```

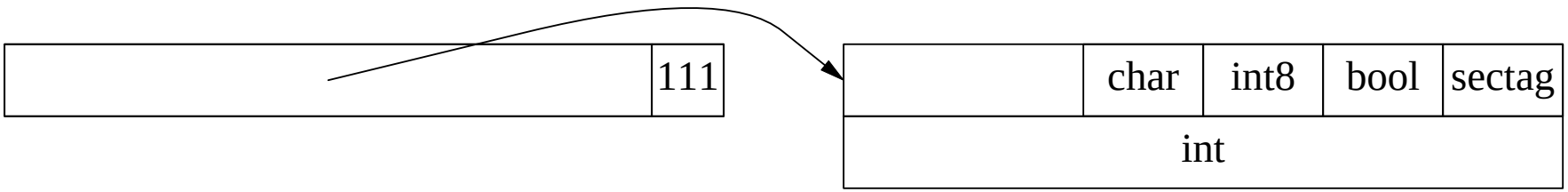
We can compare `oct(b1, b0, b1)` with `oct(b0, b1, b1)` all at once for the same reason why we can compare `101` with `011` all at once: both conditions are satisfied.

# New mixed-endian allocation algorithm

When we pack several sub-word-sized arguments totaling *N* bits into one word,

- we still allocate the absolute least significant bits either to a primary tag and a local secondary tag, or to a remote secondary tag, if needed,
- we still store the arguments in the least significant *N* available bits remaining after that, but
- when processing the arguments first-to-last, we now allocate not the *least* but the *most* significant bits still available among these.

```
:- type t ---> .... ; ... ; f(char, int8, bool, int).
```



# Compare in bulk

When an argument word contains two or more sub-word-sized arguments packed together, and all of them compare as unsigned, we can now compare the argument words from `X` and `Y` in their entirety.

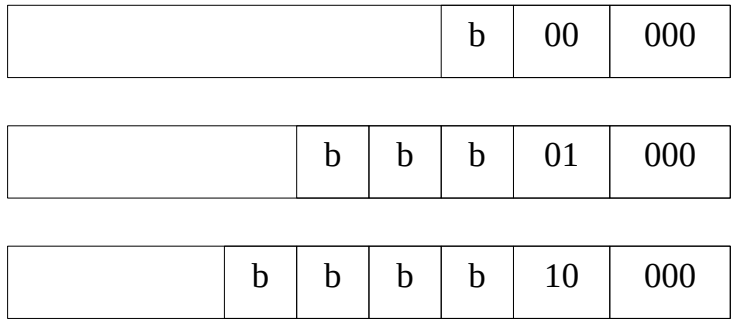
When some of the packed-together arguments compare as signed, this would not work, because it would say that e.g. -5i8 (1111 1011) is greater than 6i8 (0000 0110).

Instead, we divide the word into a sequence of segments, and compare the segments in turn. Each segment is a bitfield containing either

- one argument that compares as signed, or
- a maximal sequence of one or more consecutive arguments that compare as unsigned.

# Unify terms as a whole

```
:- type b ---> b0 ; b1.

:- type digit
    --->    binary(b).
    ;       octal(b, b, b).
    ;       hex(b, b, b, b).
```

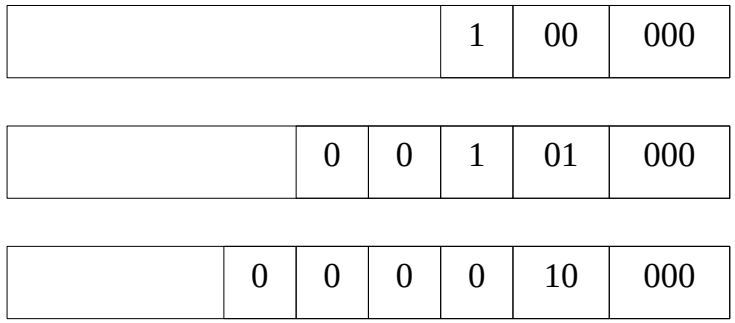



If the representation of all the functors in a type fits in one word, then we can test whether two values of that type are the same or not by testing whether the two words as a whole are equal.

# Compare term as a whole

Even if the representation of all the functors in a type fits in one word, testing two values of that type for order by testing the two words for order is *not* guaranteed to work.

For example, both of the first two words would come out as greater than the third, even though all `hex` terms should compare as greater than all `binary` and all `octal` terms.




The root of the problem is that the local sectag is stored in less significant bits than the arguments.

# Compare term as a whole

Nevertheless, compare-word-as-a-whole *does* work in two special cases:

- The type has just one function symbol with args that fit into a word.
- The type has just one constant, and one function symbol with args (that fit into a word), *in that order*.

```
:- type hex                             % special case 1
    --->    hex(b, b, b, b).

:- type maybehex                        % special case 2
    --->    null
    ;       hex(b, b, b, b).
```

The second case works because the constant will be represented by all zeroes, while (due to having ptag=1) the nonconstant cannot be all zeroes, so it *will* be recognized as being greater.

# Performance

Packing data allows bigger data sets to be handled, and makes data caches more effective.

It does take more code to access packed fields, but the cost of this code *should* be less than the cost of the cache misses that the packing avoids.

The compiler's new ability to unify, compare, copy and update packed arguments in bulk should lead to speedups as well (though bulk copy and update have been implemented only in hlc grades).

Unfortunately, due to the limitations of my work laptop, I don't have any reliable performance numbers.

# Conclusion

The hardest part of implementing all this was extending the runtime type information (RTTI) data structures to allow the description of the new argument packing schemes, and getting the runtime system to handle them correctly.

The resulting RTTI system works, but is not clean. Some parts of it would benefit from a redesign, but this would require nontrivial bootstrapping.

The biggest challenge for the future work (automatic argument reordering and the optimization of mini types) is revising the system of interface files to make it possible to fit them in.

**Argument packing is currently off by default, but now it needs only a bit more testing before being switched on by default.**

# Packed words: why MLDS/hlc only?

For the deconstruction, we generate this, and then a later pass optimizes away the unneeded assignments.

```
MR_Unsigned packed_word_0 = field(mktag(0), T0, 0);
A = (field(mktag(0), T0, 0) >> ...) & ...;
B = (field(mktag(0), T0, 0) >> ...) & ...;
C = field(mktag(0), T0, 0) & ...;
D = field(mktag(0), T0, 1);
MR_Unsigned packed_word_1 = field(mktag(0), T0, 2);
E = (field(mktag(0), T0, 2) >> ...) & ...;
F = (field(mktag(0), T0, 2) >> ...) & ...;
G = field(mktag(0), T0, 2) & ...;
```

This would be much harder to do for the LLDS: whether we want to e.g. allocate a stack slot for `packed_word_0` to hold its value across calls depends on whether and when it is used.